Science of Computer Programming 77 (2012) 990–1005

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Banzai+Tatoo: Using cutting-edge parsers for implementing high-performance servers

Julien Cervelle [a,*], Rémi Forax [b], Gautier Loyauté [b], Gilles Roussel [b]

[a] Université Paris-Est, Laboratoire d'Algorithmique, Complexité et Logique, 94010 Créteil cedex, France
[b] Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, CNRS UMR-8049, 77454 Marne-la-Vallée cedex 2, France

ARTICLE INFO

Article history:
Available online 2 March 2011

Keywords:
Software engineering
Parsing
Protocol
Server
Non-blocking IO

ABSTRACT

This paper presents how the Tatoo parser generator enables the implementation of Java high-performance servers using the Banzai generic server shell. The performance of these servers relies on the ability of Tatoo to produce push non-blocking parsers with a fixed memory footprint during parsing and on the generic and efficient server architecture of Banzai. This approach reconciles the use of formally defined grammars for protocol parsing and the efficiency of the implementation. We argue that the use of the formal grammars simplifies the implementation of the protocol and we show that an HTTP server built using the Banzai+Tatoo is as efficient as several existing specially tuned high-performance HTTP servers.

1. Introduction

Nowadays, standalone applications are found less and less frequently whereas web or network connected applications are becoming the rule. To support this evolution, high-performance servers accepting and serving a large amount of simultaneous connections have to be developed. Usually the communication protocols used by these servers are formally defined, but are implemented by hand. Heavily used tools stemming from compiler technology could be used to automate the generation of parsers for these protocols, based on their formal grammar specification. This approach would have the advantage of simplifying implementation, maintenance and the evolution of the protocol but in practice it is not used. We believe that this is due to the absence of a tool designed to support this approach and providing good performance as compared to hand-written implementations.

In this paper we propose using Tatoo Java parser generator together with the Banzai Java generic server shell to reach this goal. The Tatoo parser generator permits the simplification and enhancement of the reliability of the specification of text protocols using formal grammar definition. Banzai implements the low-level network and concurrency details. Basically, Banzai accepts incoming connections to the server and then provides high-level access to the obtained connection. For the developer, everything behaves in Banzai as if it had direct and exclusive access to the connection and processor. However, to provide high-performance IO, Banzai mixes non-blocking Java input–output [1] using selectors with pooled multi-threading considered to be the best architecture for obtaining good performance [2] in Java.

The Banzai generic server shell could be used with all types of protocol parsers (text or binary); and could therefore be used as a base for almost any kind of protocol recognition implementation. However, the protocol implementation is very complex as it requires non-blocking access to input/output which is considered difficult in practice and prone to error. This

* Corresponding author.
E-mail addresses: julien.cervelle@univ-paris-est.fr (J. Cervelle), remi.forax@univ-paris-est.fr (R. Forax), gautier.loyaute@univ-paris-est.fr (G. Loyauté), gilles.roussel@univ-paris-est.fr (G. Roussel).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.01.005

is why Banzai has been designed mostly to use parsers that are generated from a formal grammar describing a plain-text protocol using the Tatoo [3] parser generator.

Tatoo produces parsers that can operate on non-blocking input allowing several parsers to share a common thread of execution. It lowers the footprint made by the thread creations when the server is heavily loaded. In other words, instead of waiting for data to be available by blocking the current thread, like traditional parsers do, Tatoo parsers are only active when data are pushed to the parser and inactive the rest of the time. Thanks to the Java selector mechanism, Banzai activates the parsers only when data are available, simulating concurrent parsing in a single thread. Moreover, thanks to the Tatoo branching mechanism, several parsers may be combined dynamically thereby enables, for instance, the retrieval of binary or XML data during the protocol parsing.

Another important feature of the parsers produced by Tatoo, is their fixed memory consumption. These parsers can be embedded in long-life applications such as servers without requiring garbage collection. All parsers can be created at the server startup and reset after the end of each request to serve a new request. This avoids the overhead of object creation and garbage collection. Finally, the parser table used internally by the parser is shared between all parser instances recognizing the same protocol, lowering the memory footprint of the parsers running concurrently in the server.

The simplicity of our approach is comparable to XML-RPC [4] or SOAP [5] approaches on the server side. It masks the network low-level details from the developer and relies on a formally defined parsing process involving the use of a text protocol that is easy to trace and debug. However, Banzai+Tatoo is not restricted to protocols built on top of HTTP and on the verbose XML formatting and generic parsing since it also focuses on performance. The Banzai+Tatoo approach is also compatible with the REST (Representational State Transfer) [6] model that involves the use of stateless HTTP-like protocols.

In order to prove the effectiveness of this approach a simple HTTP server was developed. Its performance is compared to existing HTTP servers such as Apache httpd [7] or Grizzly [8]. The comparison with the latter shows that an HTTP server built using a formal language is as efficient as servers especially designed to provide high-performance HTTP service.

This paper is organized as follows: Section 2 uses examples to show how one can implement an efficient server with Tatoo and Banzai. Section 3 presents the interesting features of Tatoo that enable efficient parsers to be embedded in servers. Section 4 details the architecture of the Banzai generic server shell. Section 5 presents some benchmarks for comparison of the performance of the HTTP server built using Banzai+Tatoo with other servers and finally, we offer related works and conclusions.

2. Examples

In this section, we give two examples of protocol specification and implementation using the Banzai server. The first is a toy protocol example that implements a simple calendar service and is presented to illustrate the complete development process using Tatoo and Banzai. The second is a real protocol, Apple Push Notification Service [9], used to perform push notifications between iPhone application providers and Apple servers. It provides an example of a protocol where context-free parsing is required showing that the full expressive power of the parser is sometimes required. The implementation of the HTTP protocol is not presented here because of its complexity, even if the grammar specification is comparable to the calendar service protocol.

2.1. The calendar protocol example and the server development process

The protocol is based on the SMTP/HTTP request–response model [10,11]: consisting of a request line followed by several headers in the RFC822 format [12]. The request line specifies an action to be performed on a calendar located by its name, a user name for the authentication, and a protocol version. Next, the password and the requested dates are provided in several header fields. The general format of the request is illustrated by the following example:

```
GET mycalendar foo CAL/1.0
password: AjxHKRFkRwxx3j9lM2HMow==
from: Sun Nov  6 08:49:37 2008
to: Mon Nov  7 12:34:31 2008
```

This request authenticates the user `foo` and tries to retrieve the content of a calendar named `mycalendar` from November 6th to November 7th using the protocol `CAL/1.0`.

First, the developer has to specify the formal grammar of the protocol in the Tatoo format. This specification is composed of two main parts. The lexer which specifies the tokens to be recognized and those to be ignored. The following example presents this part of the Tatoo file.

```
tokens:
  service= 'GET'
  uri= '[^ ]+'
  user= '[a-zA-Z]+'
  protocol= 'CAL/1.0'
  colon= ':'
```

```
  key= '[^ :\r\n]+'
  value= '[^ \r\n][^\r\n]*'
  eoln= '\r?\n'
blanks:
  space= '[ \t]+'
```

The section `tokens` defines rules recognized by the lexer and sent as token to the parser. The section `blanks` defines rules recognized by the lexer that are not transmitted to the parser. Next, the general organization of the protocol has to be described in the grammar. The following example illustrates this second part of the file.

```
productions:
 start = request+ ;
 request = firstline eoln header* eoln ;
 firstline = service uri user protocol ;
 header = key colon value eoln ;
```

This grammar formally specifies the protocol: the possibility of sending one or more (+) requests over the same connection, the content of the mandatory first line, the possible presence of several header lines (*) and the existence of an empty line at the end of the request. This grammar specification, together with the tokens and the blanks specifications describe precisely the protocol. The implementation of complex protocols, like complex languages, may require, for the purpose of simplification, the use of semantics to precisely check the syntax. In this example the decoding of the `uri` is not treated at the lexing or parsing level but is left to the semantics. Yet, it should be noted that this protocol can be defined using simple regular expressions. The expressive power of a context-free grammar is not required here.

Writing the same protocol parser by hand would require at least several hundred lines of code and complex checks to verify the protocol details. Compared with fifteen lines of specification, it is clear that using a parser generator to implement the protocol parsing is simpler, less error-prone and much easier to maintain. Moreover, this specification is completely independent of the underlying implementation. The developer can concentrate on the protocol and does not have to bother with low-level details of input–output, particularly complex to cope with in the context of non-blocking implementation, following the "separation of concerns" technique [13].

2.1.1. Parser generation

From the previous formal description, the Tatoo parser generator called with proper options produces several Java files for the parsing and lexing of the protocol. It also produces the glue code that allows the lexer and the parser produced by Tatoo to be embedded into the Banzai generic server shell.

This glue code is encapsulated into a `ProtocolHandler` abstract class. The developer has to extend this abstract class to implement the "semantics" (behavior) of the server during protocol parsing. More precisely, first the developer has to implement the special behavior of the server when the lexer recognizes a new token (implementing the `tokenRecognized()` method). It can decide to retrieve the token value and to perform complementary checks. Second, it has to implement the higher-level semantics, when a production has been recognized[1] by the parser (implementing the `productionRecognized()` method). Moreover, the end of a request sent by the client is associated with a particular production. Complex actions, such as file retrieval and communication back to the client, have to be implemented at this level. This is why the `ProtocolHandler` abstract class also provides an API for interacting with the client connection through Banzai: for sending data (stored into large memory buffers managed by Banzai) back to the client in an asynchronous way (`pipelineWriter.asyncWrite()`) or for terminating the connection.[2]

External services, such as authentication or access to a complex file cache, may be provided to the semantics at the time of server creation.

2.1.2. Example of semantics implementation

The first part of the following `CalendarProtocolHandler` class implements the decoding of our calendar protocol, extending the `ProtocolHandler` abstract class described above.

```
public class CalendarProtocolHandler
       extends ProtocolHandler {
  private String uriValue; private String username;
  private String key; private String value;
  private HashMap<String,String> headerMap=...

  public void tokenRecognized(RuleEnum rule,
```

[1] The productions are recognized upward since Tatoo produces bottom-up parsers.

[2] By default, the connections are kept alive and accept more than one request.

```
                    TerminalEnum terminal) {
  switch(terminal) {
    case uri:   uriValue=decode(); return;
    case user:  userName=decode(); return;
    case key:   key=decode(); return;
    case value: value=decode(); return;
  }
}
public void productionRecognized(ProductionEnum production) {
  switch(production) {
    case header:  headerMap.put(key,value); return;
    case request: handleRequest(); return;
  }
}
```

Tatoo provides two APIs to get information from the parser. A low-level API which encodes the tokens and the productions used in the grammar specification as Java enumerations[3] and provides access to the lexer buffer. A higher-level API, implemented on top of the low-level API, encodes each production as a Java method[4] taking as argument the attribute value computed from the recognized token or an upward method call. We decided to base Banzai API on the low-level API and to let the developers use their own data structures. This decision was made since high-level API requires to maintain another stack of attributes[5] which may not be needed when parsing protocols that do not require the full power of context-free grammars.

The inherited `tokenRecognized()` and `productionRecognized()` methods respectively receive as argument the token and the production enumeration value. This dispatch could have been implemented using attribute grammars, but separating the grammar from the semantics was one of the early design choices in Tatoo driven by the separation of concerns principle.

The `decode()` method, provided through Banzai, enables the retrieval of the string value of the tokens. Note that, the token values are only retrieved and decoded when the string value is useful. For instance, even if the method `tokenRecognized()` for both the `protocol` token (`CAL/1.0`) and the `uri` token, only the `uri` string value is decoded.

In the same way, productions that are not useful for the semantics are not taken into account in the `productionRecognized()` method, e.g. the `firstline` production. On the contrary, `request` production is important when it is found, because it means that the complete request from the client has been recognized. In this example, the developer provides the protocol semantics of a request using the method sketched below:

```
private void handleRequest() {
  try {
    String password=headerMap.get("password");
    boolean passwdOK=authenticationService.
      checkAccess(username,password,uriValue);
    if (!passwdOK) {
      putUnauthorizedResponse(outBuffer);
      pipelineWriter.asyncWrite(outBuffer);
      return;
    }
    ...
    Calendar c=...
    putCalendar(outBuffer,c);
    pipelineWriter.asyncWrite(outBuffer);
  } catch(IOException e) {
    pipelineWriter.endConnection(e);
  }
}
```

Apart from authentication,[6] the above method performs data output. Several methods and fields are provided to the developer to access the socket connection managed at a lower level by Banzai. In particular, Banzai provides a large buffer to write data, and a pipeline that permits asynchronous writings of the buffer to the socket connection or in order to close the connection.

[3] If the grammar changes these enumerations may change as well and the class has to be modified and recompiled.

[4] This API is similar to Yacc, the method parameter are equivalent to $i and return value to $$.

[5] In fact, two stacks because objects and primitive values do not have a common supertype in Java.

[6] If the `password` is not available in the headers a `null` value is extracted from the header map and the authentication fails.

When authentication fails, a special response is returned to the client before the connection is closed by Banzai. Note that if an error occurs during the parsing of the request, Banzai closes the connection without an intermediate response. In the same way, un-handled Java exceptions thrown during the execution of the semantics are caught by Banzai and the connection is closed. On the contrary, if the client does not send data or does not close the connection, Banzai closes the connection based on a selector timeout.

2.1.3. Server construction

Once the protocol handler is implemented, it has to be plugged into the Banzai generic server shell. This is done by overriding the `createHandler()` method of the parser pool class (`ParserRequestAnalyzerPool`).

Parser objects and their semantics handlers are then created when the pool is constructed at the server startup. The pool is in charge of recycling the parser objects between requests.

```
ParserRequestAnalyzerPool pool =
  new ParserRequestAnalyzerPool() {
    @Override
    protected ProtocolHandler createHandler() {
     return new CalendarProtocolHandler(authServ);
    }
  };
```

The user then creates the server and starts it. At that moment, it is also possible to specify the listening port and the number of threads for each task of the server as described in Section 4.

```
BanzaiServer server=
  new BanzaiServer(port,pool,readerThreadCount,
                   parserThreadCount,writerThreadCount);
server.start();
```

2.2. Apple push notification service grammar example

This second example illustrates the need to have a full-featured parser able to recognize context-free text-based protocols. Indeed, the protocol for Apple Push Notification Service, even if messages are currently limited to 293 bytes, cannot be parsed using simple regular expressions and finite state automaton. A grammar description as well as the power of a stack automaton is needed.

The APNS protocol is described by the following grammar; the lexer part is omitted here:

```
productions:
  start = header payload ;
  header = 37bytes ;
  payload = object ;
  object = lbrace field/,+ rbrace ;
  field = string : value ;
  value =  string | int | double | object |
           array | true | false | null ;
  array = lsquarebracket value/,* rsquarebracket ;
```

In this example, the Tatoo notation `element/sep+` means one or more `element` separated by `sep` and `element/sep*` means zero or some `element` separated by `sep`.

This protocol cannot be recognized by regular expressions since nested `objects` and `arrays` may occur. Thus, it is necessary to recognize nested braces and square brackets, which requires the power of context-free grammars.

3. Tatoo

Most of the optimizations of the servers produced using Banzai+Tatoo can be done thanks to features available in the Tatoo parser generator, introduced in our previous work [3,14], and implemented in Java.

The main feature that enables efficient protocol parsing is the generation of push parsers[7] allowing the server to use non-blocking IO.

3.1. Basic features of Tatoo

The next section briefly introduces pull parsing, the following sections describe push parsing and other useful features of Tatoo.

[7] Even if push and pull parser terminology is commonly used for XML parsers, this terminology is also applicable for traditional parsers.

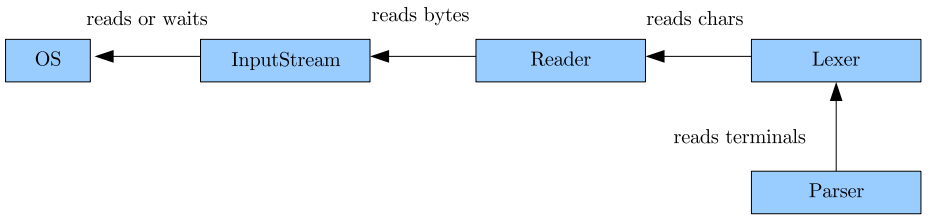

Fig. 1. Pull parser and lexer.

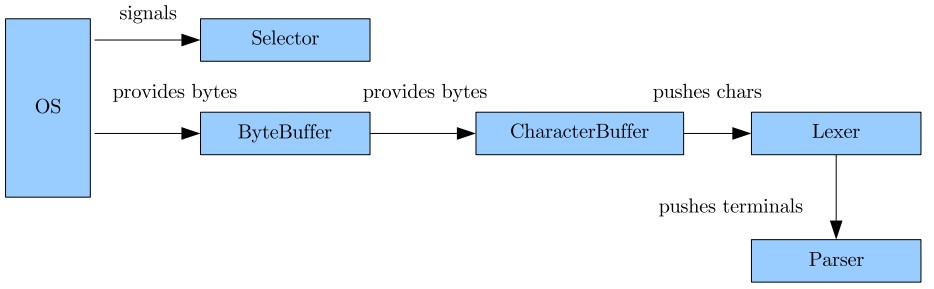

Fig. 2. Push parser and lexer.

3.1.1. *Pull parsing*

Traditional parser generators such as Yacc [15] or ANTLR [16] use pull parsers. The principle of pull parsing is illustrated in Fig. 1.

The pull parser controls the whole parsing process: this process blocks the current thread, waiting for input (even if the lexer is decoupled from the parser) and proceeds when new data or tokens are available, until the end of the parsing. This process requires the use of blocking IO because the state of the parser is maintained by the parsing function call stack, meaning that one thread can only parse one client request at a time. Indeed, it is not possible[8] to suspend the function call, to save the call stack, to start another parser and later to resume the first parser with its initial call stack. When data are not available for one parser, the process switches to another thread using the thread context switching mechanism provided by the OS or the VM. This approach is not effective because it causes many system-level context switches. Such switching is particularly penalizing when data arrive in small blocks of bytes or in heavy loads, where a large number of concurrent requests is used, system performance can decrease because of the high number of threads to manage [17]. This must be avoided in high-performance servers to resist a flooding of requests.

3.1.2. *Push parsing*

Conversely, a Tatoo parser does not wait for data but, when data are available, a buffer containing this data is fed (pushed) into the parser (using the method `step()`). When all available data in the buffer are processed, the parser saves its internal state and returns. This kind of implementation is closely related to reactive programming [18] and relies heavily on the NIO API [1] introduced in Java 1.4. More precisely, it relies on mechanisms such as NIO selectors to inform the parser of the presence of data on input and on non-blocking read to only retrieve available data without blocking. The general push parsing process is illustrated in Fig. 2.

With this mechanism, one thread can serve several clients "simultaneously". Indeed, several parsers are managed in the thread, each of them ready to parse data provided by one client. The Java NIO selector mechanism is then used to wait simultaneously for data on the active client connections. When data are available on several connections, the corresponding parsers are activated/resumed one after another, using the `step()` method. In other words, using the calendar example, if the selector detects that `G` is available on one connection and `GET` on a second, then `G` is pushed to a first parser and when it stops, `GET` is pushed to the second parser. If, in the meanwhile, `ET` arrives on the first connection, the first parser is resumed, etc.

3.1.3. *Memory management*

The second important feature of the generated parsers is their memory management which has been specially tuned for long living and in order to lower the memory footprint in the presence of concurrent parsing. This is not the case for the traditional parser implementations usually designed for singlerun off-line parsing.

This means that the parsing state for each connection is stored in a parser object. This object may be reused after the end of a request without the help of the Java garbage collector. It only contains the stack and the current state of the parsing and it shares a unique LR table with all of the parser objects that run concurrently.

[8] At least until JVM fully supports continuations.

The number of parsers is established at server startup. The parser, the lexer and the table objects are fix-sized objects; only the stack memory footprint of the parser may increase during the parsing. Moreover, instead of transforming all byte sequences into Java `String` objects, character encoding attached to the parsers at the time of creation allows them to work directly on the input byte sequence. This avoids potential character decoding and the construction of unnecessary `String` objects.

All of the objects that are not created at server startup are recycled by the Java garbage collector. This strategy ensures that space consumption is bounded.

3.1.4. Rule selection

Another important feature of Tatoo parsers is that lexing rules are selected according to what the parser expects in the current state. The lexer does not try to recognize tokens that are not expected by the parser, thereby eliminating unnecessary computation. For instance, at the beginning of the parsing of an HTTP request, the parser only expects the token `GET` or the token `POST`. Hence, only the two corresponding lexing rules are active and any request that does not start with a `G` or a `P` is immediately rejected by the lexer.

The direct implication is that, when a character is pushed to the lexer, only a small number of the automata is updated, which improves the performance of the analysis. This is shown in [19] for general purpose languages (C#, Java, etc.) and domain-specific languages [20] (protocols can be considered as special kinds of DSL) that the average number of active automata only represents between one sixth and one half of the total number of available rules. Moreover, during the profiling of the Tatoo analyzers embedded in Banzai, profilers show that the analyzer spends most of the time in the code that runs automata. Hence, only the lexing is less efficient than keeping a parser and a lexer with rule selection enabled.

Another consequence of rule selection is that all parsing errors are treated as soon as possible since parsing errors are detected at the lexer level. Because the parser instructs the lexer what tokens to expect, only the corresponding lexer rules are active. If an unexpected terminal occurs, the associated lexer rule is not active and a lexer error is raised [3].

3.1.5. More features

In addition to the above features, Tatoo adds several others which are quite convenient for protocol parsing.

- Grammars can be specified with versions attached to productions, to easily support different protocol versions (like HTTP/0.9, 1.0 and 1.1). The versions are embedded in the same parser table. Thus, the same parser can be reused dynamically by different clients that use different protocol versions.
- The semantic actions are triggered as soon as possible. For instance, Tatoo performs reductions in LR0 states without waiting for one more look-ahead token.
- The computation of the semantic values associated with the tokens is performed only when it is needed (that is, when the corresponding terminal is shifted, it would use too much memory to wait for the reduction). This allows more flexibility for error recovery and branching (branching is explained in the next section), and saves computation time.

3.2. Theoretical issues

Since our parser generator produces bottom-up parsers, the parsing is done in linear time, the memory used (for the parsing) per client is the input buffers and the parser stack. The latter is quite negligible compared to the former for most of the protocols since the height of the syntax tree is either constant (for protocols which are regular languages) or logarithmic in the size of the whole message, provided the grammar is well written. Since the message is processed by chunks, the size of the buffer is bounded by the size of the longest token.[9] Note that in Banzai+Tatoo, memory security is not an issue since, as it is written in Java, no buffer overflow can occur.

In benchmarks (Section 5), Banzai+Tatoo is either comparable to or better than servers written in the "traditional way". The reasons why hand-written protocol parsing may not be optimal could be:

- that the use of LL parsers with function calls are difficult to combine with push parsing and use of thread pools
- the need to have the whole message or the whole header ahead of time in order to start the parsing since it is almost impossible to foresee all of the possible ways of cutting the header into chunks efficiently (note that this is done automatically in Tatoo with a parser stack)
- the need to have the whole message or header before rejecting or closing a connection.

3.3. Tatoo branching mechanism

Since version 3.0 of Tatoo, the runtime allows sub-parsers to be dynamically called from some statically identified *branching* non-terminals associated with a specific sub-grammar. This feature enables a grammar to have pluggable parts

[9] If the buffer is smaller than the currently parsed token, it is automatically extended or rejected if the size has reached a user-defined threshold.

without running multiple parsers in parallel. It allows language extension and selection between different grammars at runtime depending on some semantics values. The implementation of parsers is generated independently and is linked dynamically.[10] The runtime then efficiently but not exactly (as described below) apes the behavior of a static parser generated from a grammar obtained by unification of the branching non-terminals with the axioms of the branched grammar.

We initially designed the branching mechanism for the separate compilation of grammars [14] and later found it to be particularly interesting for protocol parsing. Indeed, it is common to find during parsing some "objects" embedded into the messages of the protocol. For instance, in SOAP, the representation of an XML method call may be embedded into the HTTP request or, more basically, binary data may be embedded in HTTP GET requests.

The basic idea of this mechanism is to make sure that when the parser reaches a branching point, the parser is stopped and some other mechanism (like another parser) processes the input until the control returns to the original parser. This requires three specific behaviors for the parsers:

- that they do not consume input even if they use some look-ahead during the parsing
- that they support a stop and resume mechanism
- that they are able to detect, at runtime, that the current parser state accepts potential branching points.

Tatoo push parsers are well suited to support separate compilation because implementing a push parser also requires implementing the first two behaviors. However, the initial algorithm was too simple to support all of the specificities required by protocol parsing. This is why we developed a new algorithm, described in this section.

3.3.1. Initial branching algorithm

The initial algorithm [14] works as follows. Branching non-terminals are treated for LR table generation as terminals. During runtime, the first parser is run and if a parsing error occurs and a branch to a sub-parser is possible, the corresponding sub-parser is started.

This algorithm has the advantage of being very simple but the drawback of this behavior that is not equivalent to parsing using the regular grammar where the branching non-terminals are unified with the axioms of the branched grammars. More precisely, the difference is that, when the parsing is correct, the sub-parser is never called, which is sometimes not the desired behavior. Indeed, if the first parser accepts at a branching point the same token as the sub-parser in the initial state, the sub-parser could be chosen instead of the first parser after disambiguation. The ambiguity may also occur at the lexer level. For instance, if one uses the following grammar for expressions where the *atom* is declared as a branching non-terminal:

$$E \quad \rightarrow \quad E + E \mid E - E \mid E/E \mid E \times E \mid \ -E \mid (E) \mid id(E) \mid atom$$

where the following grammar is branched:

$$A \quad \rightarrow \quad id \mid int \mid float.$$

The second grammar can never use production $A \rightarrow id$ because the production $E \rightarrow id(E)$ is used instead, since the first grammar is of a higher priority.

Though a static analysis could detect these undesired behaviors, there is sometimes no easy way to remove them using the initial algorithm.

3.3.2. The new algorithm

The major modification of the new algorithm is to run lexers concurrently from both the initial grammar and the branched grammar, and to allow the specification of policies to choose, when branching can be performed, between entering the sub-parser or continuing with the regular parsing process. The decision is based on information available at that time. For instance, in the previous example, one could choose to continue the parsing (with production $E \rightarrow id(E)$) if the recognized identifier is a function in the symbol table and to branch (with production $A \rightarrow id$) otherwise. Note that this approach is still weaker than combining the grammars at compile time since the parser and the sub-parser are not run in parallel for better performance and in order to avoid the need for undoing semantic actions.

From an implementation point of view, a policy object responsible for the branching decision is provided, just before a branch, with all of the tokens that, according to the input, could be shifted. These tokens may come from both the branched grammar and from the initial grammar since both lexers are run concurrently. This extension adds flexibility to the branching mechanism. It allows the developer to have full control over the inclusions. By extension, it also allows binary content to be embedded into the text protocol.

In order to implement this extension, we use two of the special features of Tatoo. The first consists in delaying, for as long as possible, the semantic actions (in particular the computation of the semantic value of tokens) in order to be able to run several lexers concurrently and to be able to cancel some of them. The second is the possibility to tune the lexer so that it uses a selected set of possible rules associated with tokens potentially coming from different grammars.

[10] If the branching is known at the start of the parsing, every computation necessary to perform branching can be done statically. However, it is pointless to do it at compile time since, in this case, it would be better to unify the grammars and construct a single parser table.

3.3.3. Detailed description of the algorithm

In this section we detail the algorithm used to select the set of lexing rules active in the branching points and how the branched parsers are started and stopped. In this description, the *branching non-terminals* are abbreviated: "bnt". We say that a terminal is "*shiftable*" in a parser if it can be shifted, possibly after some reduce actions. This property can be tested in constant time since this means that the corresponding action in the current parser's state is not an error.

Branching chain

In a branching point, when one particular rule of the lexers succeeds and is selected, the parser has to decide from the produced terminal which branching action (to branch, to continue the parsing, etc.) has to be performed. For this purpose, a *branching chain* is associated with the terminal. This chain is a sequence composed of *enter* and *leave* actions. More precisely:

- Enter grammar G from a bnt B of a grammar H means:
  - G is branched to a bnt B of H,
  - and that B is shiftable, in the parser for H.

  If this enter action is performed, the parser for H is paused and a parser for G is initiated.
- Leave grammar G back to H means:
  - a bnt of H is branched to G and the parsing has entered grammar G from H,
  - the end-of-input token is shiftable, in the parser for G.

  If this leave action is performed, the parser for G terminates and the parser for H is resumed.

The objective of the first algorithm is to compute the set of terminals (and their rules) associated with a shiftable branching non-terminal, but without performing the branching. For this purpose, two procedures are defined.

Procedure ENTER($\mathcal{P}$, c): From a parser $\mathcal{P}$ that is reached following chain c, compute the set of the terminals that are shiftable just after an enter action

Given a branching chain c and a parsing $\mathcal{P}$ in a state where a bnt B is shiftable then, for each such B, if G is the grammar branched to B, then let d be the branching chain c extended with "enter G from B":

- for each terminal t in First(S), where S is the axiom of G, output t associated with d;
- if G accepts the empty word, output a special terminal ε associated with d. This fake terminal corresponds to a rule that always succeeds (consequently, it is up to the policy to choose it or not). We prefer this solution rather than adding the terminals shiftable in the state reached by $\mathcal{P}$ after having shifted B. Indeed, this could lead to inefficient computation in the event of multiple grammar branching, for instance if an end-of-input token or another bnt is found;
- recursively call ENTER($\mathcal{Q}$, d) where $\mathcal{Q}$ is the parser of the grammar G in its initial state (it will do nothing if no bnt are in First(S)).

Procedure LEAVE($\mathcal{P}$, c): From a parser $\mathcal{P}$ reached following chain c, computing the set of terminals that are shiftable just after a leave action

Given a branching chain c, and parsing in $\mathcal{P}$ in a state where the end-of-input terminal is shiftable, if $\mathcal{P}$ was initiated after branching from parser $\mathcal{Q}$, then let d be the branching chain c extended with "leave" (no need to put "to $\mathcal{Q}$" since it is unique):

- for all shiftable terminals t in $\mathcal{Q}$, output t associated with d;
- recursively call ENTER($\mathcal{Q}$, d) and LEAVE($\mathcal{Q}$, d) (this will add terminals if $\mathcal{Q}$ reaches an accept state or can shift some bnt).

To compute the set of terminals and the branching chains, these two procedures are called with the current parser following empty chain. Adding to this set, the terminals shiftable by the current parser associated with an empty chain, we obtain a set of lexing rules (the regular expressions associated with each terminals) used by the lexer to recognize tokens in the branching points.

Conflicts may occur during this construction: for instance, given B_1 and B_2 two branch terminals, if $G_1 : S \to B_1|B_2$ and both bnt are branched to the same grammar then the same terminal is associated with several branching chains. This kind of conflict is resolved by a policy implemented by the user (the default behavior is priority based).

From an implementation point of view, the two procedures must be implemented with dynamic programming to avoid going into infinite loops. For instance, if one procedure is recursively called for the same parser (not the same instance but a parser in the same state on the same grammar), the procedure must return immediately outputting nothing. Since the sets are implemented efficiently using binary tables and thanks to dynamic programming, these algorithms are linear in the number of grammars plus the number of bnt (the first items of the procedures are constant time). Moreover, note that, initially, the sets may be pre-computed if the branching is known before starting the parsing (however, it should be noted that Tatoo allows dynamic branching) and second, these algorithms can run in constant time if the user avoids sequences of bnts and the presence of bnt in the First set of axioms.

Let us illustrate the above algorithm with an example. Suppose we have the following grammars of axiom S, where B_i are bnts branched to G_i,
$G_1 : S \rightarrow$ 'z' | B_2 B_3 | B_2 'u'
$G_2 : S \rightarrow$ 'x' | 'x' B_1 'y' | B_3
$G_3 : S \rightarrow \varepsilon$ | 't'.

Suppose the initial grammar is G_1, the chains computed in its initial state are (remember that ε corresponds to a fake terminal):

'z'	empty chain
'x'	enter G_2 from B_2
ε, 't'	enter G_2 from B_2, enter G_3 from B_3.

Suppose 'x' is chosen, we push it to a new parser for grammar G_2 which reaches state {$S \rightarrow$'x' •, S $\rightarrow$'x' • B_1 'y'}. Then, the set of terminals together with their chains is:

'u'	leave
ε, 't'	leave, enter G_3 from B_3
'z'	enter G_1 from B_1
'x'	enter G_1 from B_1, enter G_2 from B_2
ε, 't'	enter G_1 from B_1, enter G_2 from B_2, enter G_3 from B_3.

Note that there is a conflict here for ε and 't' that the user must resolve.

Performing the action

With the previously defined set of lexing rules, the lexer is run and a terminal is produced. Then, the following actions are performed.

- The branching chain associated with the chosen terminal is followed, entering and leaving grammars.
- The terminal attribute is computed.
- The terminal is pushed to the last parser of the branching chain.

More precisely, after the lexer finishes running finite automata corresponding to all of those terminals, a runtime policy that has been specified by the user selects which terminal has to be produced (using, for instance, traditional criteria like length and priority [15], or others, for example such as semantic considerations). If the terminal is associated with an empty chain, no branching is performed and the current parsing continues. If not, the actions along the branching chain are performed. At each step, we maintain a "current parser". For each action A in the chain,

- if A is "enter G from B", the parser for grammar G is created and B is pushed to the current parser, delaying the semantic action (see below). The newly created parser becomes the current parser;
- if A is "leave", the end-of-input is pushed to the current parser which causes the production of a semantic value v. Let $\mathcal{P}$ be the parser that created the current parser from a branching terminal B. The delayed semantic action of $\mathcal{P}$ is now performed by giving value v to B. Then, $\mathcal{P}$ becomes the current parser.

Delaying semantic actions means that the parser state is changed without performing any semantic action. This feature allows the parser to be in a good state to compute the shiftable terminals for the leave action and to wait for the attribute produced by the branched parser that may be useful for the semantics.

3.3.4. Binary inclusion

Using the branching mechanism, one could also plug a binary parser (for instance based on a previously specified length in the protocol as found in HTTP chunks or the header value Content-Length). For this purpose, an ad hoc binary parser has to be implemented and instantiated by the developer. Moreover, it has to provide information to the algorithms previously described.

The binary parser is solicited first to provide the active rule in the branching point, then this rule is used to determine if the input is valid, playing the role of the lexer. It should not perform any semantic action in case a token from another parser is recognized and chosen by a policy. If the policy chooses the binary token, the binary parser object is called again to read the input and to perform semantic actions.

For instance, for a length-based binary inclusion, the code of the binary parser would be the following:

```
public class LengthBinaryInclusion
      implements BinaryParser<LexerBuffer> {
  private int length;
  private final BinarySemantics semantics;

  public void setLength(int length) {
    this.length = length;
  }
```

```
  /*we do not check available length here, indeed we do not
    want to read data in advance if the branch does not
    have the highest priority*/
  @Override public boolean accepts(LexerBuffer buffer) {
    return true;
  }

  /*returns true if more input is needed*/
  @Override public boolean parse(LexerBuffer buffer) {
    if (buffer.available() < length)
      return true;
    buffer.unwind(length);
    semantics.perform(buffer);
    return false;
  }
}
```

The `accepts()` method is called by the lexer and the `parse()` method is called to perform the semantic action.

4. The Banzai architecture

In this section, we present the architecture of the Banzai generic server shell. This architecture has been designed to accept the push parsers produced by Tatoo and to take full advantage of their features. However, the global performance of the servers also relies on using non-blocking IO and a sparing use of threads.

Indeed, the implementation of a high-performance server requires that the interleaving of client requests not be penalized by data transfer latency: one client should be served when another is waiting for data. This interleaving can be performed using threads or non-blocking mechanism. We adopt a mixed strategy comparable to that of SEDA [2].

4.1. Principles

To reduce thread context switching and the use of selector when it is not necessary, the whole Banzai implementation tries to adhere to a simple principle:

"Try to be iterative i.e to perform read/write in the current thread; if this is not possible, delegate the same iterative task to a helper thread; if in the helper thread, this is still not possible, register the task in a selector."

The basic idea behind this principle is to use complex mechanisms (threads, selectors) that induce cost overrun (context switching, synchronization) only when necessary, trying to do as much as possible in a single thread in an iterative manner.

However, developing a high-performance server in Java using non-blocking IO is not an easy task mostly because developers have to focus on contradictory concerns. In particular, thread-safety and concurrency are difficult to manage in the presence of Java selectors [21]. Selectors make monitoring simultaneously several input/output possible but their implicit use of monitors is complex to manage in the presence of other monitors implied by multi-threading. Moreover, non-blocking IO requires using an event-based programming model which is harder to implement and to debug than an iterative one. Finally, objects like large input/output buffers or heavily used Tatoo parsers need to be pooled to avoid excessive garbage collection and an unnecessarily long pause time. Thus, careful memory management has to be implemented.

4.2. Detailed architecture

The architecture of the Banzai generic server shell is illustrated in Fig. 3. In this figure, the classes `ParserRequestAnalyzer`, `Parser` and `MiniToolsListener` are generated by Tatoo from the grammar specification. They depend on the developer implementation and thus are not part of Banzai. The class `Service` interacts with the class written by the developer (`ProtocolHandler`) and a `PipelineWriter` is used to ask to schedule the writing by Banzai.

The server workflow is split into different tasks. A thread pool and a scheduling policy are attached to each task, thus tasks run concurrently. The number of threads of the reader, parser and writer tasks, can be specified by the developer when the server is created (see Section 2).

The first task, the acceptor task, accepts the connections, the second one, the reader task, reads the request which may be fragmented, the third one, the parsing task, parses each fragment and delegates to a handler the protocol behavior. When requested by the handler, the writer task, writes the response in one or more byte arrays fragments back to the client. Each task is connected to the next one using round-robin queues that evenly dispatch the work between the threads.

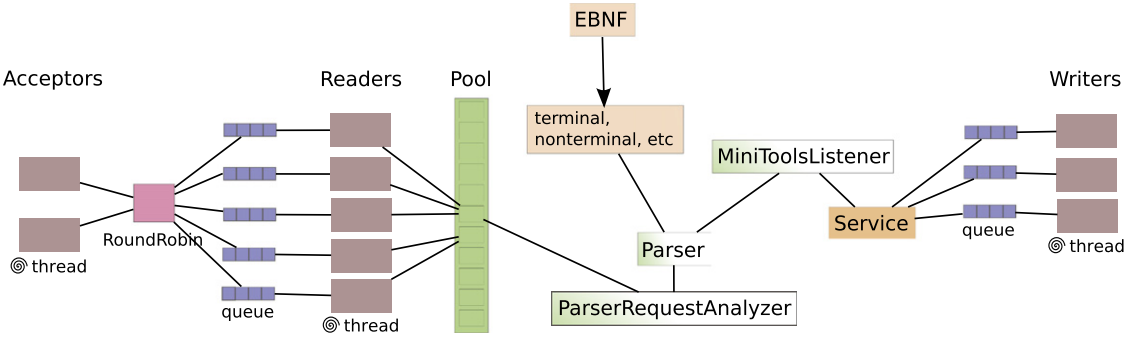

Fig. 3. Banzai architecture.

4.2.1. Acceptor task

Experimentally we noticed that a single thread could not accept more than 15,000 concurrent connections, thus we introduced several threads to accept connections.[11] Moreover, we noticed that because of the delay induced at TCP connection time by the three-way handshake, no or little data are available at that time. Thus, contrary to our global principle, no read action is performed just after accepting a connection since it usually returns no data. Each acceptor dispatches the client socket to one of the reader threads using a simple round-robin mechanism and one queue per reader.

4.2.2. Reader task

Reader threads wait on the selector associated with each reader task until at least one registered socket has some data to be read or is prompted by an acceptor thread because a client socket is available in a concurrent queue.

In the two cases, the thread tries to read data from the client socket. If data are available, it calls the parsing task with the data, otherwise it registers the client socket in the selector to wait for data availability.

4.2.3. Parsing task

The first time a request requires parsing, the parsing task gets a `ParserRequestAnalyzer` from a pool (see Section 2.1.3). This object contains a Tatoo parser and a protocol handler. Since, the state of the parsing is encapsulated into the parser object, it cannot be shared between multiple socket connection. Thus, one `ParserRequestAnalyzer` object is "attached" to each socket connection. This object will be returned to the pool when the socket connection to the client is closed. Each time data are available on the connection, the `ParserRequestAnalyzer` is retrieved and, inversely, when data are written by the protocol handler, the connection is retrieved.

The parsing is handled by the Tatoo parser that processes all data read and invokes the methods of the service. It calls the method `tokenRecognized()` each time a token is recognized and `productionRecognized()` each time a production is applied. The protocol handler stores the transformed data like the decoded URL. It also interacts with the client using the methods of the `PipelineWriter`: `asyncWrite()` to send a buffer to be written back to the client and `endConnection()` if the connection must be closed.

4.2.4. Writer task

Following our principle, when the protocol handler wants to write response data, it first tries to write the data in the parsing thread, if data cannot be written, it delegates data chunks to a writer thread. All data written will be sent to the same writer thread to preserve serialization of the data. When a protocol handler asks to close the connection using `PipelineWriter.endConnection()`, the connection will be closed only after all data are written.

5. Benchmarks

In order to assess the efficiency of the Banzai+Tatoo, we implemented an HTTP server HTTPBanzai and compared its performance to existing HTTP servers.

The ApacheBench [22] and Httperf [23] tools were used to evaluate the number of requests that can be served per second during predefined, but varying, load scenarios. ApacheBench only tests HTTP version 1.0.

All measurements were done with a server running on a two-way SMP 2.4 GHz Pentium IV system with 2 GB of RAM and Linux 2.6.19, Sun JDK 1.7.0 beta 24 as the Java platform. Two machines with a similar configuration were used for the load generation. All machines were interconnected using a Gigabit Ethernet switch.[12]

[11] Two threads are used in our benchmarks since we realized that the server could not treat more than 30,000 simultaneous connections (see Section 5).

[12] This configuration does not simulate wide-area network effects but it is sufficient since our concern is with the performance of the servers under heavy load.

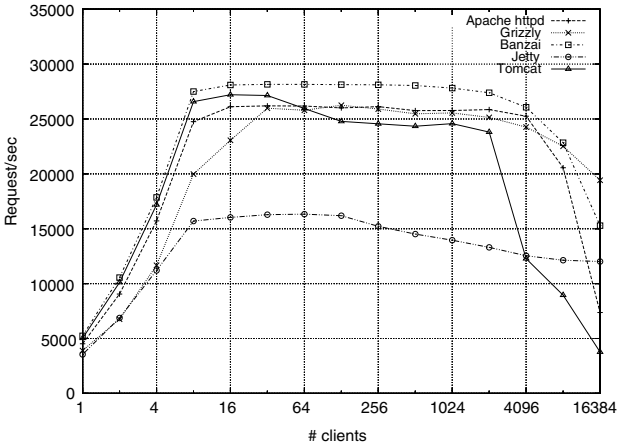

Fig. 4. Served requests per second, given a certain number of clients, with ApacheBench.

For the purpose of comparison, the popular Apache httpd [7], written in C, as well as the Grizzly [8], Jetty [24] and Tomcat NIO [25] Java web servers were tested. Apache is a process-based web server, where each client connection is handled entirely by a single process. Grizzly, Jetty and Tomcat NIO are based on threads and on the Java NIO API.

The HTTPBanzai server is very basic since it only serves static files even if it performs all of the required checks in this context. One could argue that our comparison is unfair since the other servers provide more complex services such as dynamic pages generation. However, our tests only concern access to static pages under heavy load and thus, the effect of their complexity is canceled by the redundancy of client requests: requests are directly served from the cache in all servers. Moreover, this work focuses on protocol parsing and low-level server performance. The use of higher-level mechanisms such as dynamic page generation during the benchmark would probably disrupt the results.

Because we want to compare server frontend (Network IO and protocol parsing) performance without being disrupted by disk IO, all measures are done after a warm-up that loads the requested file into the cache.

Fig. 4 presents the number of requests served per second, given a certain number of clients for the different servers done with ApacheBench. For Banzai, this number increases concurrently with the number of clients up to 8 clients. This is normal behavior for a server in HTTP 1.0. The maximum number of requests that the server could handle is then reached, that is to say about 28,000 requests per second. The server maintains the same performance until 4096 clients access the server. In the end, the limits of the system are reached and the performance drops. The HTTPBanzai server behaves differently in excess of 4096 clients compared to the Grizzly server since the HTTPBanzai server does not close persistent connections even in the presence of heavy load. To circumvent this variation in performance, a mechanism that statistically closes connections under heavy load without waiting for timeout, has to be implemented. We considered it as beyond the scope of this work.

Except in the case when the limits of the system are reached, the performance of HTTPBanzai is always better than that of the other servers. This is particularly true when the number of clients exceeds 32 and the number of requests served per second is very stable.

The good performance of HTTPBanzai is perhaps due to its simplicity, even if the HTTP grammar is complete and the server performs numerous checks. It could also be due to the fact that the protocol handler does not wait for the end of the request to start the treatment (see Section 3.2). In particular, it retrieves the requested file "descriptor" as soon as the URL is available, taking advantage of connection latency, if the end of the request is still not available. More precisely, the sequence of actions performed by the server is not the following: *accept a connection, parse the request, do the treatment, send back the response and close the connection*, the parsing of the request action and parts of the treatment action may overlap.[13]

The same test, performed with `httperf`, is presented in Fig. 5. The performance of the different servers is relatively stable. As in previous tests, the performance goes down in excess of 4096 clients. The better performance of HTTPBanzai seems to be due to the heavy use of HTTP1.1 persistence that reduces the connection overhead. No error is reported by `httperf`.

A second type of test, presented in Fig. 6, shows the evolution of the number of requests given the size of the files served. Here again the performance of HTTPBanzai is better than that of the other servers up to a common limitation of the system that applies to all servers. Tests done with ApacheBench and Httperf are comparable.

In order to trace the memory of Grizzly and HTTPBanzai, the behavior of both servers was captured with `jconsole` using the same ApacheBench test. The memory consumption of BanzaiHTTP is presented in Fig. 7 and Grizzly's in Fig. 8. During the test, memory consumption of HTTPBanzai remains limited below 10 Mb but grows until garbage collection occurs. Since the parsers and buffers used are pooled and reused, the increase in heap memory usage is probably due to the fact that the Java network API creates several small objects for each network connection and HTTPBanzai uses a few strings to decode

[13] For other protocols, these actions may also overlap with the writing action.

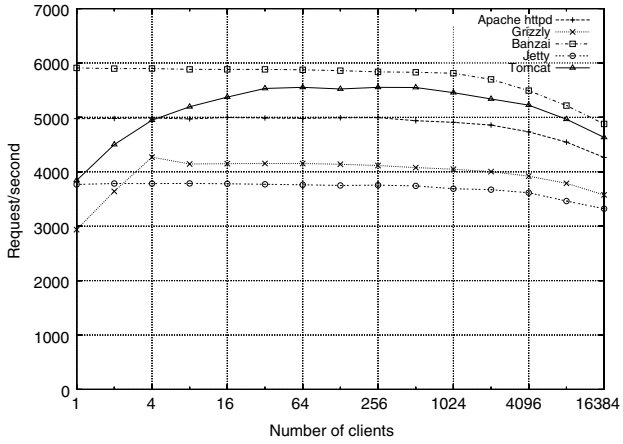

Fig. 5. Served requests per second, given a certain number of clients, with `httperf`.

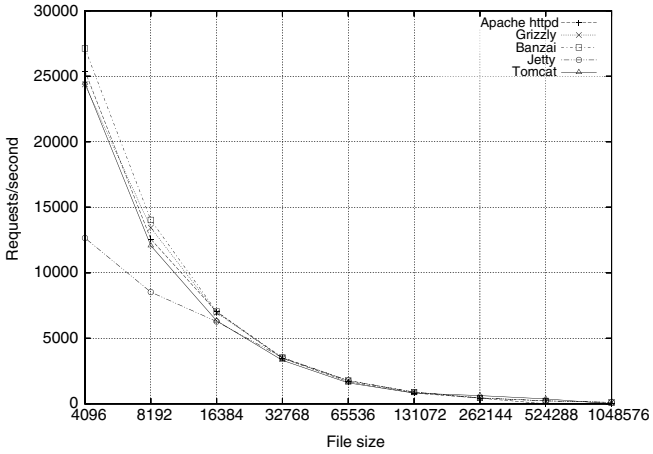

Fig. 6. Served requests per second, given the size of the file done with ApacheBench.

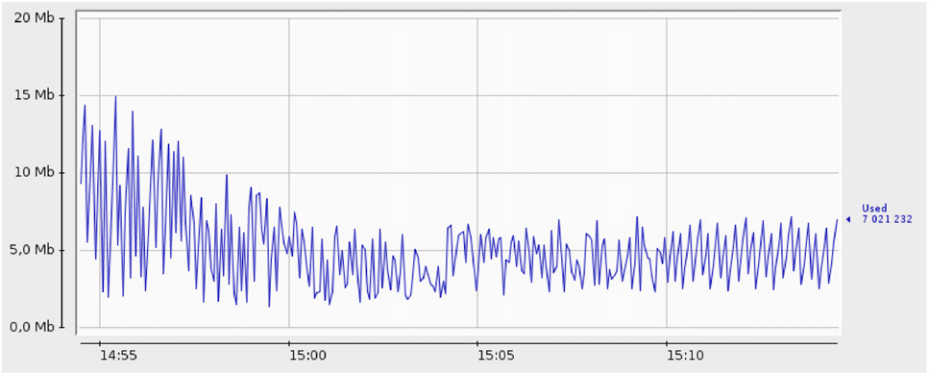

Fig. 7. Memory consumption of HTTPBanzai.

part of the request (URI, connection state, etc.). Grizzly also uses pooled buffers but the parsing is done using dynamically allocated buffers of characters and strings leading to more memory consumption.

6. Related work

Parser generators for ASCII and binary protocols such as DATASCRIPT [26], PADS [27], APG [28], binpac [29], GAPA [30] and Zebu [31] were developed to address the growing complexity of the network protocol parsing. These tools use domain-specific language or traditional grammars to describe the protocol. Some of them are especially designed to parse binary

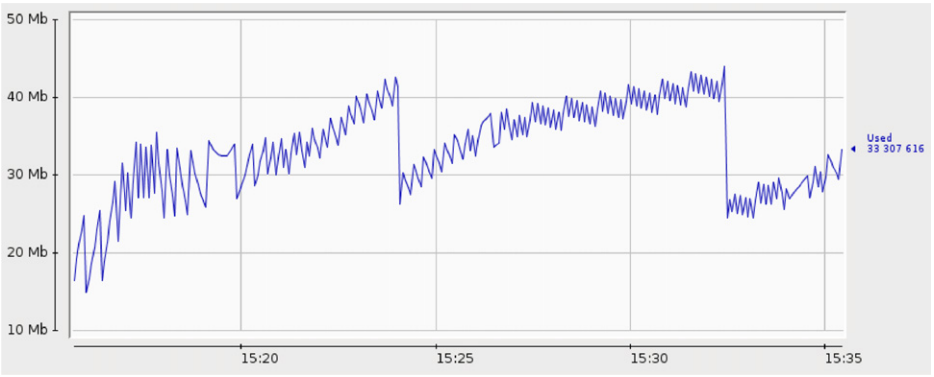

Fig. 8. Memory consumption of Grizzly.

protocols, which is not the case of Tatoo and they are available for languages other than Java. However, they do not produce push parsers and thus cannot be used efficiently in a non-blocking environment. Moreover, they do not propose a generic high-performance server designed to embed the produced parsers.

Packet types [32] is a tool which helps convert packets into a programming language data structure, performing easy packet type detection and automatic byte-order conversion between network and host integers. It is also intended to be used in BSD kernel. Even if Tatoo's binary element handling is basic, we provide an interface for custom code. Transposing the code of Packet types on top of Tatoo's binary parser interface could be an interesting future work.

Several projects such as Tuxedo [33], JAWS [34,35], SEDA [2], Twisted [36] and Serveez [37] supply libraries that provide the features necessary to quickly develop Web servers. These frameworks focus on server architecture and are usually more flexible than Banzai+Tatoo. However none of these also focus on protocol parsing.

MSPL [38] and NEST [39], similar to the Banzai+Tatoo, combine the formal specification of the protocol with the complete generation of server implementation. In MSPL, a compiler is used to generate the low-level implementation of communication for both the client and the server side from a declarative description of the protocol. One of the drawbacks of this approach is that it does not give access to the underlying implementation language. In NEST, a specification close to Lex and Yacc enables the generation of the server code. One of the principal characteristics of this approach is the possibility to generate three types of servers (multi-process, multi-threaded and event-driven), which is not the case for Tatoo where only event-driven is available since it is the most efficient. NEST is limited to the Linux platform and, as it is written in C, allows buffer overflow errors and has no runtime optimization when compiling the code. This is well suited to long-running application like servers.

Peake and Salzman, propose a modularization of grammars [40] based on an object-oriented extension mechanism. Like the Tatoo extension mechanism, they propose separate compilations of the grammars. The implementation is based on LL parsing that simply allows the productions implemented with functions to be overwritten, replacing the original production by the new overwriting one (together with the associated semantics).

Silver [41] is a recent parser generator which shares features with Tatoo-like selecting rules from the parsing context and branch grammar. The branching algorithm, though it is different, is akin in power to the initial branching algorithm of Tatoo without support for nested branches (a grammar is branched to itself or in loops). Silver's branching algorithms perform statical analysis to ensure that no runtime problem can occur. We did not implement such checks in Tatoo since many users of our first branching algorithm found the gap between the power of grammar composed at compile time and this kind of runtime branching algorithm was too great.

7. Conclusion

In this paper we show that using the Tatoo parser generator to recognize protocols is a simple and effective way to produce text-based servers. Indeed, Tatoo generates powerful push parsers that support non-blocking IO and low memory consumption that can be embedded into the efficient and generic server architecture of Banzai. Moreover, Tatoo's branching mechanism simplifies the specification of protocols that embed some binary data or other protocol chunks. A basic HTTP server implemented using this approach displays a better performance than other existing HTTP servers, proving its usability.

In the future, we would like to include the Banzai architecture into the Saburo [42] framework to provide high-performance servers in this context. Another future work will consist in providing a framework that is based on a formal definition, not only for parsing requests but also for the generation of the responses.

We also want to simplify the implementation of the protocol handler which still requires solid knowledge of Tatoo, perhaps by introducing attribute grammar notation.

References

[1] New I/O APIs, 2002, http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html.
[2] M. Welsh, D.E. Culler, E.A. Brewer, SEDA: an architecture for well-conditioned, scalable Internet services, in: Proc. of the 8th Symposium on Operating Systems Principles, Chateau Lake Louise, Canada, 2001, pp. 230–243.
[3] J. Cervelle, R. Forax, G. Roussel, Tatoo: an innovative Parser Generator, in: Proc. of the 4th Conference on Principles and Practice of Programming in Java, Mannheim, Germany, 2006, pp. 13–20.
[4] D. Winer, XML-RPC, http://www.xmlrpc.com/spec.
[5] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H.F. Nielsen, A. Karmarkar, Y. Lafon, Soap version 1.2, http://www.w3.org/TR/soap12.
[6] R.T. Fielding, Architectural styles and the design of network-based software architectures, Ph.D. Thesis, University of California, Irvine, 2000.
[7] Apache Software Foundation, The Apache HTTP Server Project, http://httpd.apache.org.
[8] Sun Microsystems, GlassFish / Grizzly Project, https://grizzly.dev.java.net.
[9] Apple Inc., Apple push notification service, 2009, URL: http://developer.apple.com/iphone/library/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/Introduction/Introduction.html.
[10] J. Postel, Simple Mail Transfer Protocol, RFC 821 (Standard), obsoleted by RFC 2821, Aug. 1982, URL: http://www.ietf.org/rfc/rfc821.txt.
[11] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext transfer protocol – HTTP/1.1, RFC 2616 (Draft Standard), updated by RFC 2817, Jun. 1999, URL: http://www.ietf.org/rfc/rfc2616.txt.
[12] D. Crocker, Standard for the format of ARPA Internet text messages, RFC 822 (Standard), obsoleted by RFC 2822, updated by RFCs 1123, 2156, 1327, 1138, 1148, Aug. 1982, URL: http://www.ietf.org/rfc/rfc822.txt.
[13] W. Hürsch, C. Lopes, Separation of concerns, Tech. Rep. NU-CCS-95-03, College of Computer Science, Northeastern University, Feb. 1995.
[14] J. Cervelle, R. Forax, G. Roussel, A simple implementation of grammar libraries, Computer Science and Information Systems 4 (2) (2007) 65–77.
[15] S. C. Johnson, Yacc: yet another compiler compiler, in: UNIX Programmer's Manual, vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 353–387.
[16] T.J. Parr, R.W. Quong, ANTLR: a predicated-LL(k) parser generator, Software Practice and Experience 25 (7) (1995) 789–810.
[17] J. Ousterhout, Why threads are a bad idea (for most purposes), in: Proceedings of the USENIX Annual Technical Conference, San Diego, CA, USA, 1996, invited talk.
[18] D. Harel, A. Pnueli, On the development of reactive systems, Logic and Models of Concurrent Systems 13 (1985) 477–498.
[19] M. Črepinšek, T. Kosar, M. Mernik, J. Cervelle, R. Forax, T. Kosar, G. Roussel, On automata and language based grammar metrics, Computer Science and Information Systems 7 (2) (2010) 309–329.
[20] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344.
[21] Sun Microsystems, New I/O APIs, http://java.sun.com/j2se/1.4.2/docs/guide/nio.
[22] Apache HTTP server benchmarking tool, http://httpd.apache.org/docs/2.0/programs/ab.html.
[23] Hewlett Packard, Httperf, http://www.hpl.hp.com/research/linux/httperf.
[24] Mort Bay Consulting, Jetty, http://www.mortbay.org.
[25] Apache Tomcat 6.0, http://tomcat.apache.org/tomcat-6.0-doc/index.html.
[26] G. Back, Datascript: a specification and scripting language for binary data, in: Proc. of the 1st Conference on Generative Programming and Component Engineering, Springer-Verlag, 2002, pp. 66–77.
[27] K. Fisher, R. Gruber, PADS: a domain-specific language for processing ad hoc data, SIGPLAN Notices 40 (6) (2005) 295–304.
[28] D.T. Lowell, APG: an ABNF Parser Generator, Jun. 2006, http://www.coasttocoastresearch.com.
[29] R. Pang, V. Paxson, R. Sommer, L. Peterson, Binpac: a Yacc for writing application protocol parsers, in: Proc. of the 6th Conference on Internet Measurement, Rio de Janeriro, Brazil, 2006, pp. 289–300.
[30] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, C. Guo, A generic application-level protocol analyzer and its language, in: Proc. of the 14th Annual Network and Distributed System Security Symposium, 2007.
[31] L. Burgy, L. Reveillere, J.L. Lawall, G. Muller, A language-based approach for improving the robustness of network application protocol implementations, in: Proc. of the 26th International Symposium on Reliable Distributed Systems, Washington, USA, 2007, pp. 149–160.
[32] P.J. McCann, S. Chandra, Packet types: abstract specification of network protocol messages, SIGCOMM Comput. Commun. Rev. 30 (4) (2000) 321–333. http://doi.acm.org/10.1145/347057.347563.
[33] BEA, Tuxedo, white papers. http://www.bea.com.
[34] J.C. Hu, D.C. Schmidt, Developing flexible and high-performance Web servers with frameworks and patterns, ACM Computing Surveys 32 (1) (2000) 39–45.
[35] J. Hu, D.C. Schmidt, JAWS: a framework for high-performance web servers, in: Domain-Specific Application Frameworks: Frameworks Experience by Industry, John Wiley and Sons Ltd, 1999, pp. 339–376.
[36] G. Lefkowitz, I. Shtull-Trauring, Network programming for the rest of us, in: Proc. of the USENIX Annual Technical Conference, San Antonio, USA, 2003, pp. 77–89.
[37] S. Jahn, Serveez, http://www.gnu.org/software/serveez/manual/index.html.
[38] M.A.L. Douglas, Mspl: a protocol language for generating client–server software, Ph.D. Thesis, Florida Institute of Technology, May 2000.
[39] K. Wilson, J. Aycock, NEST: network server tool, in: 11th Asia-Pacific Conference on Communications, 2005, pp. 1107–1111.
[40] I. Peake, E. Salzman, Support for modular parsing in software reengineering, in: STEP'97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice, STEP'97 (including CASE'97), IEEE Computer Society, Washington, DC, USA, 1997, p. 58.
[41] A.C. Schwerdfeger, E.R. Van Wyk, Verifiable composition of deterministic grammars, SIGPLAN Not. 44 (6) (2009) 199–210. doi: http://doi.acm.org/10.1145/1543135.1542499.
[42] G. Loyauté, R. Forax, G. Roussel, Saburo: a tool for I/O and concurrency management in servers, in: Proc of the 20th International Parallel and Distributed Processing Symposium, Rhodes Island, Greece, 2006.